ACS Basic

User's Manual

v1.20

3/6/2009

On The Cutting Edge of Technological Evolution

6233 E. Sawgrass Rd • Sarasota, FL. 34240 • (941)377-5775 FAX(941)378-4226
www.acscontrol.com

Copyright © 2002-2009 by ACS, Sarasota, FL ALL RIGHTS RESERVED

Table of Contents

Table of Contents 2

Features 5

Programs 5

Symbolic Abbreviations 6

Variables 7

Special Variables 7

@TIMER(x) 7
@PORT(x), @PORT2(x) 7
@CONTACT(x) 7
@CLOSURE(x) 8
@OPENING(x) 8
@FEOF(#N) 8
@SECOND, @MINUTE, @HOUR, @DOW, @DATE, @MONTH, @YEAR 8
@SOUND$ 9
@VOL, @NSVOL 9
@BAUD 9
@MSG$ 10
@SOM 10
@EOM 10
@PTT 10
@MUTE 10
@LINEIN 10
@DMXRESET 10
@DMXMASTER 10
@DMXFRAMEDELAY 10
@DMXCHANNELS 10
@DMXDATA(x) 11
@DMXANALOG(x) 11
@LCDADDRESS 11
@LCDTYPE 11
@SOUNDFRAMEPRESCALER 11
@SOUNDFRAMESYNC 11

Events 12

Statements 13

CLEAR 13
CLOSE #N 13
DATA 13
DEL path 13
DELAY value 13
DIM var(size)[, ...] 13
DIR [path] 13
DIR #N, [path] 14
EDIT line 14
END 14
ERROR value 14
EXITFOR line 14

FINPUT #N, var[$], … , var[$] 14
FOR var=init TO limit [STEP increment] 14
FOPEN #N, recordlength, "path" 14
FREAD #N, recordnumber, var[$], var[$], … var[$] 15
FWRITE #N, recordnumber, var[$], var[$], ... var[$] 15
FINSERT #N, recordnumber, var[$], var[$], ... var[$] 15
FDELETE #N, recordnumber 15
GOSUB line 15
GOTO line 15
IF test THEN line/statement [ELSE line2/statement2] 15
INPUT var 16
INPUT "prompt", var 16
INPUT #N, var 16
[LET]var[$]=expr[$] (default statement) 16
LIF test THEN statement[:statement] 16
LIST [start, [end]] 16
LIST #N [start, [end]] 16
LOAD path 17
MD path 17
MEMORY 17
NEW 17
NEXT [var] 17
ON expr, GOSUB line0, line1, line2, … ,lineN 18
ON expr, GOTO line0, line1, line2, … , lineN 18
ONERROR GOTO line 19
ONEVENT @specialvar, GOSUB line 19
OPEN #N, "path", "options" 21
ORDER line 21
PLAY file 21
PRINT expr[, expr ...] 21
PRINT#N, expr[, expr ...] 21
READ var[,var ...] 21
RETURN 21
REM 21
REN oldfile, newfile 22
RUN [line] 22
SAVE [path] 22
SIGNAL @specialvar 22
STOP 22
TYPE path 22
WAIT @specialvar 22
LCDx Statements 23
LCDPRINT row[s], col, font, type, justify, expr (@LCDTYPE=0) 23
LCDPRINT rowstart, col, font, type, justify, expr (@LCDTYPE=1) 23
LCDUNPRINT row[s], col, font, type, justify, expr (@LCDTYPE=0) 23
LCDUNPRINT rowstart, col, font, type, justify, expr (@LCDTYPE=1) 23
LCDCLEAR row[s], colstart, colend (@LCDTYPE=0) 23
LCDCLEAR rowstart, rowend, colstart, colend (@LCDTYPE=1) 23
LCDGRAPHIC row[s], col, data (@LCDTYPE=0 only) 23
LCDLINE startx, starty, endx, endy, color 23
LCDBOX corner1x, corner1y, corner2x, corner2y, color 23
LCDPIXEL x, y, color 23
LCDCIRCLE x, y, radius, color 23
LCDTONE frequency, duration 23
LCDSAVE page 24
LCDRESTORE page 24
LCDBITMAP startrow, col, "path" 24

Operators **25**

Expressions **26**

Functions **26**

ASC(char) 26
ABS(expr) 26
CHR$(expr) 26
ERR( ) 26
ERR$( ) 26
FMT$(fmt$, expr[$]) 27
MID$(var$, start, len) 28
LEFT$(var$, len) 28
RIGHT$(var$, len) 28
INSERT$(var$, start, var2$) 28
REPLACE$(var$, start, var2$) 28
LEN(var$) 28
FIND(var$, searchvar$) 28
RND(expr) 29
VAL(expr$) 29
STR$(expr) 29
MULDIV(number, multiplier, divisor) 29
MULMOD(number, multiplier, divisor) 29

Examples **31**

Setting the Real Time Clock 31
Two Sound Sequences 32
Activating Multiple Output Contacts for a Sound 33
Control from a Serial Port 34
Westminster Chimes 35
Jukebox with Display 38
Fixed Length Record File I/O 42
Error Logging 43
DMX Control Synchronized to Sound 44

Firmware Revisions **46**

NOTICE: **48**

Features

ACS Basic is an integer, microcomputer basic designed for simple control applications.

ACS Basic executes programs consisting of one or more statements. Statements consist of an optional line number followed by reserved keyword commands specifying operations for Basic to perform followed by required and / or optional arguments.

Statements that begin with a line number are entered and held, sorted by line number, until Basic is commanded to execute them. This is called the **Program** mode of operation. Statements entered without a line number are evaluated and executed immediately. This is called the **Direct** mode of operation. Some keyword commands are **Direct** mode only and may not appear in a program. Some keyword commands are **Program** mode only and may not be evaluated and executed immediately after being typed in. These limitations are listed in the keyword command definitions below.

Programs

In ACS Basic a Program consists of one or more program lines. Each program line consists of a line number followed by one or more statements. Multiple statements in a program line must be separated by colons (":"). Program lines that are entered without a line number are executed directly. Only certain statements may be executed directly. When ACS Basic is awaiting statement or program line entry it issues a READY prompt via the serial port.

```
ACS Basic v1.4 Sep 25 2006 11:44:00
Ready
dir *.bas
TEVENT.BAS          250 A      11-09-2058 14:30:10
PROGRAM1.BAS         55 A      11-09-2058 15:52:44
SOUNDS.BAS          248 A      01-01-1980 00:00:00
TEST.BAS             63 A      01-01-1980 00:00:00
CEVENTS.BAS         144 A      01-01-1980 00:00:00
PROGRAM2.BAS         47 A      11-09-2058 15:58:14
ONGOTO.BAS          253 A      05-08-2052 14:35:54
ONGOSUB.BAS         272 A      11-09-2058 14:45:08
TIMER.BAS           185 A      11-15-2058 15:20:26
CHIMES.BAS          884 A      09-07-2021 16:55:10
LCDDEMO.BAS        2143 A      11-13-2020 18:36:26
MSGTEST.BAS          78 A      11-11-2020 16:15:32
----------------------
                   12 files
                    0 directories
Ready
```

Programs may be entered a line at a time by a stream of characters via the serial port, or by loading from a file off of an optional Compact Flash card. When entered via the serial port, a program line will replace any matching program line, and entering a line number only will delete the corresponding program line. Entered program lines are limited to 127 characters of length.

```
    10 PRINT "This is a Test"
    20 STOP
    list
    10 PRINT "This is a Test"
    20 STOP
    Ready
    20
    list
    10 PRINT "This is a Test"
    Ready
    run
    This is a Test
    Ready
    print "This is also a Test"
    This is also a Test
    Ready
```

ACS strongly recommends developing Basic programs interactively via a connected terminal / computer so that error messages can be viewed and the program operation can be refined quickly – otherwise the program may silently stop running leaving no clue as to what has happened.

Program lines may be viewed with the **LIST** statement. All program lines may be cleared with the **NEW** statement. Program execution is started using the **RUN** statement. Upon power-up, ACS Basic clears the program memory and awaits statement or program line entry via the serial port.

Program lines may be edited via a connected ANSI terminal with the **EDIT** statement. (See the EDIT keyword command definition below for more information.)

Entering an Escape character (0x1B) twice in succession via the serial port while a program is running will cause termination of the program and ACS Basic will output a message then await further statement or program line entry via the serial port. If the program is awaiting input by executing an INPUT statement a trailing carriage return may be necessary to terminate the INPUT before the Escape sequence is seen.

```
new
Ready
10 for i=1 to 10
20 print i
30 delay(10)
40 next i
list
10 FOR i=1 TO 10
20 PRINT i
30 DELAY(10)
40 NEXT i
Ready
run
 1
 2
 3
 4    <- Escape key pressed twice here
ESC at line 20
Ready
```

Symbolic Abbreviations

In the reserved keyword and command definitions to follow, the following symbolic abbreviations apply:

#N	Represents a file number #**0** <= **#N** <= **#9**
var	Represents a numeric program variable
var$	Represents a string program variable
var()	Represents a numeric array program variable
@specialvar	Represents a special program variable
line	Represents a program line number **0** <= **line** <= **65535**
[ ]	Delineates optional arguments or parameters
path	Represents a complete path to a file including the filename without leading backslash. There is no concept of a current directory other than the root filesystem.
statement	Represents a program statement
expr	Represents a program expression
recordlength	Represents a Fixed Length File I/O record length **0** <= **recordlength** <= **127** including the trailing CR/LF on the end of each record
recordnumber	Represents a Fixed Length File I/O record number **0** <= **recordnumber** <= **32767**

Variables

ACS Basic has three types of variables: **16-bit Integer Numeric**, **16-bit Integer Numeric Arrays** and **unsigned 8-bit character Strings**. Variable names *are not* case sensitive.

Numeric variables can assume the integer values **(–32768 < variable < +32767)**. Character Strings are limited to **127 characters** in length.

The 260 Numeric variables are named **A0 – A9 … Z0 – Z9**.

The 260 Numeric Array variables are named **A0( ) – A9( ) … Z0( ) – Z9( )**. Array variables must be DIMensioned using the DIM statement before use.

The 260 Character variables are named **A0$ - A9$ … Z0$ - Z9$**.

Note that the zero suffix variables may be referenced by their letter name only so that A is equivalent to A0, Z$ is equivalent to Z0$, etc.

Special Variables

ACS Basic also has built-in special variables. Special variables are denoted by a '**@**' character as the first character of the variable name. The special variable names are 'tokenized' when entered to save program memory and speed program execution: for example the special variable **@SECOND** would be tokenized to two bytes instead of seven bytes.

Special variables **may not** be assigned a value by appearing in an **FOR, DIM, INPUT, READ, FINPUT #N** or **FREAD #N** statement. Some special variables are read-only and may not appear on the left hand side of a **LET** assignment statement.

Some special variables have ***Events*** associated with them and may be referenced in **ONEVENT, SIGNAL** and **WAIT** statements. See the description for the individual special variables and the ***Events*** section below for more information.

@TIMER(x)

The **@TIMER(x)** special variables allow the ACS Basic program to measure or control time intervals. There are ten timers; permissible values for (x) are 0 through 9. Setting the variable to a non-zero value activates the timer. The value in the timer variable is decremented every 20mSEC (50 Hz) until it reaches zero. Upon reaching zero any associated event handler specified with the **ONEVENT** statement is activated.

@PORT(x), @PORT2(x)

The **@PORT(x)** and **@PORT2(x)** special variables allow the ACS Basic program to access I/O ports. There are 256 eight bit ports; permissible values for (x) are 0 through 255. Setting the variable to a value writes the value to the I/O port (x). Reading the variable returns the value from the I/O port (x). Note that ports 0, 1 and 2 are consumed by optional installed CFSound-3 Contact I/O modules.

@CONTACT(x)

The **@CONTACT(x)** special variables allow the ACS Basic program to access CFSound-3 contacts. There are up to 56 contact inputs and up to 16 contact outputs depending upon what optional modules have been installed on the CFSound-3; permissible values for (x) are 0 through 55. Setting the variable to a '1' activates output contact (x). Reading the variable returns the value from the input contact (x).

@CLOSURE(x)

The **@CLOSURE(x)** special variables allow the ACS Basic program to access CFSound-3 contact events. . There are up to 56 contact inputs depending upon what optional modules have been installed on the CFSound-3; permissible values for (**x**) are 0 through 55. Reading the variable returns a '1' if the input ***contact(x)*** has had a closure since last being read. The program must 'clear' the closure by assigning a zero before it can be detected again. Optionally an event handler specified with the **ONEVENT** statement may be activated upon in input closure, which automatically clears the closure.

```
10 ONEVENT @CLOSURE(24),GOSUB 100
20 ONEVENT @CLOSURE(25),GOSUB 200
30 GOTO 30
100 PRINT "contact 25 closed":RETURN
200 PRINT "contact 26 closed":RETURN
Ready
run
contact 25 closed
contact 26 closed
 ESC at line 30
Ready
```

@OPENING(x)

The **@OPENING(x)** special variables allow the ACS Basic program to access CFSound-3 contact events. There are up to 56 contact inputs depending upon what optional modules have been installed on the CFSound-3; permissible values for (**x**) are 0 through 55. Reading the variable returns a '1' if the input ***contact(x)*** has had an opening since last being read. The program must 'clear' the opening by assigning a zero before it can be detected again. Optionally an event handler specified with the **ONEVENT** statement may be activated upon in input opening, which automatically clears the opening.

@FEOF(#N)

The **@FEOF(#N)** special variable allows the ACS Basic program to determine when an end-of-file has occurred after an **INPUT #N, FREAD #N** or **FINPUT #N** statement. Optionally an event handler specified with the **ONEVENT** statement may be activated upon an end-of-file occurring.

@SECOND, @MINUTE, @HOUR, @DOW, @DATE, @MONTH, @YEAR

These special variables allow the ACS Basic program to access the Real-Time Clock/Calendar. Writing one of these variables except @SECOND stops the clock and updates the associated value. Writing to the @SECOND variable updates the value and starts the clock running. The values of these variables are updated once per second. Whenever one of the values of these variables changes, any associated event handler specified with the **ONEVENT** statement is activated. See the ***Setting the Real Time Clock*** sample program in the Examples section for more information.

@SECOND	00 <= **seconds** <= 59
@MINUTE	00 <= **minutes** <= 59
@HOUR	00 <= **hour** <= 23
@DOW	0 <= **day of week** <= 6 (read-only, 0=Sunday)
@DATE	1 <= **date of month** <= 31
@MONTH	1 <= **month of year** <= 12
@YEAR	00 <= **year** <= 99

@SOUND$

The **@SOUND$** special variable allow the ACS Basic program to queue sound files for playing. Queued sound files are played in the order that they were queued. A sound is queued by assigning the string value of the sound filename to the variable. The currently playing sound may be determined by reading the value of the variable. The queue may be flushed by assigning an empty string to the variable. When the queue becomes empty any associated event handler specified with the **ONEVENT** statement is activated.

@VOL, @NSVOL

The **@VOL** and **@NSVOL** special variables allow the ACS Basic program to control the CFSound-3 volume. The volume is set by assigning a numeric value to the variable. The current volume may be determined by reading the numeric value of the variable. The range is 0 (mute) to 63 (max volume). Note that the **@VOL** volume setting is saved in non-volatile memory and is restored every time the CFSound-3 powers up. ***The non-volatile memory has a limited number of write cycles (~100,000) and can be worn out by excessive writes so this function should not be used in a loop and with caution.*** The @NSVOL volume setting doesn't save the value in the non-volatile memory and doesn't have a use limit, however the volume will be restored to the last @VOL or pushbutton set value upon the next power-up or reset.

@BAUD

The **@BAUD** special variables allow the ACS Basic program to control the CFSound-3 serial port baud rate. The baud rate is set by assigning a numeric selector value to the variable. The current baud rate selector may be determined by reading the numeric value of the variable. A selector is used to allow baud rates greater than 28800 which would result from the 16-bit integer limitation of the Basic language. Note that the baud rate selector is saved in non-volatile memory and is restored every time the CFSound-3 powers up. ***The non-volatile memory has a limited number of write cycles (~100,000) and can be worn out by excessive writes so this special variable should not be written in a loop or on every program execution. Exercise caution to avoid non-volatile memory failure. A good practice is to check the variable's value and only then write to it if it is not the desired value.***

@BAUD	Baud Rate
0	110
1	300
2	600
3	1200
4	1800
5	2400 (factory default)
6	3600
7	4800
8	7200
9	9600
10	14400
11	19200
12	28800
13	38400
14	57600
15	115200
16	230400

@MSG$

This special variable is updated by receipt of a serial data stream message that is framed with the **@SOM** and **@EOM** characters which are not included in the **@MSG$**. It retains the framed message until it is read at which point the search for the next received **@SOM** begins again. It may also be cleared by assigning it a string value, which is not saved.

@SOM

This special variable determines the character used to delineate the Start of Message. It defaults to ASCII SOH (01).

@EOM

This special variable determines the character used to delineate the End of Message. It defaults to ASCII ETX (03).

@PTT

Writing this special variable to a non-zero value activates the PTT relay. Setting it to zero deactivates the PTT relay. Reading this special variable returns 1 if the PTT relay is active, else zero.

@MUTE

Writing this special variable to a non-zero value mutes the speaker amplifier. Setting it to a zero value un-mutes the amplifier. Reading this special variable returns 1 if the amplifier is muted, else zero. The RUN command automatically un-mutes the speaker amplifier.

@LINEIN

Writing this special variable to a non-zero value enables the Line level Input. Setting it to zero disables the Line level input. Reading this special variable returns 1 if the line level input is enabled, else zero. The RUN command automatically disables the Line level input. Audio on the Line level Input is amplified to the current volume level and is presented to the speakers and Line level Output when it is enabled and no other sound is playing.

@DMXRESET

Writing this special variable to a non-zero value resets the optional DMX I/O module if present.

@DMXMASTER

Writing this special variable to a non-zero value enables the optional DMX I/O module as a master, controller if present. A value of zero enables sets slave, device mode.

@DMXFRAMEDELAY

Writing this special variable sets the inter-frame delay in multiples of 20mSEC when the optional DMX I/O module is present and configured as a master.

@DMXCHANNELS

Writing this special variable sets the number of channels transmitted times 2 if the optional DMX I/O module is present and configured as a master.

@DMXDATA(x)

Gets or sets the current value of channel x (0 <= x <= 511) if the optional DMX I/O module is present.

@DMXANALOG(x)

Gets or sets the current value of analog input x (0 <= x <= 7) if the optional DMX I/O module is present.

@LCDADDRESS

This special variable sets the current value of the LCD address to be used with all of the LCDx commands. When a LCDx statement is processed, the value of @LCDADDRESS is tested.

If @LCDADDRESS is set to a value greater than or equal to zero, the generated LCDx commands include the LCD address prefix characters (0 <= @LCDADDRESS <= 255) inserted after the initial SOH and before the command character.

If @LCDADDRESS is set to a value less than zero, the generated LCDx commands do not include the LCD address prefix characters.

@LCDADDRESS defaults to a value of -1 when ACS Basic is started, the NEW statement is executed or a program is loaded.

See the ACS-LCD-128x64 and ACS-LCD-320x240 Display User Manuals for additional information about display addressing.

@LCDTYPE

This special variable sets the current value of the LCD type which controls the operation of the LCDx commands. The currently supported values are 0 = ACS LCD128x64 command formatting (the default), 1 = ACS LCD320x240 command formatting.

@LCDTYPE defaults to a value of 0 when ACS Basic is started, the NEW statement is executed or a program is loaded.

See the ACS-LCD-128x64 and ACS-LCD-320x240 Display User Manuals for additional information about display addressing.

@SOUNDFRAMEPRESCALER

This special variable sets the value of the number of 20mSEC (50Hz) ticks that elapse between @SOUNDFRAMESYNC events while a sound is playing.

@SOUNDFRAMESYNC

This special variable returns the current frame number of the playing sound. It starts at zero when a sound starts playing, and advances at the @SOUNDFRAMEPRESCALER rate. Due to implementation latency it can be off from 0 to 20mSEC from the actual start of the sound playing, but this offset should remain constant for the duration of the sound play out. Optionally, an event handler specified with the **ONEVENT** statement may be activated whenever @SOUNDFRAMESYNC changes. This is a 16-bit signed integer that will wrap negative as it increments past 32767 requiring a judicious choice of @SOUNDFRAMEPRESCALER value to allow the range to accommodate the length of the sound being synchronized to:

@SOUNDFRAMEPRESCALER=1 yields 20 mSEC per frame => max 655 second sound

@SOUNDFRAMEPRESCALER=50 yields 1 SEC per frame => max 32768 second sound

Events

ACS Basic provides the concept of an ***Event***. Events occur outside of the normal program execution flow and are processed in between the execution of individual program statements. Some special variables have ***Events*** associated with them and may be referenced in **ONEVENT, SIGNAL** and **WAIT** statements.

There are two ways to process an event: asynchronously with an **ONEVENT** handler or synchronously with a **WAIT** statement or by polling the special variable's value in the program.

In order to process an event asynchronously, Basic has to be informed of what code to execute when a certain event occurs. This is done using the **ONEVENT** statement. After Basic executes each program statement, it scans the table of events looking to see if any has been signaled. If an **ONEVENT** handler for a signaled event has been specified by the program, then Basic will force a subroutine call to the event handler before the next program statement is executed.

Events have an implicit priority with higher priority events being able to interrupt execution of lower priority event handlers. Here's an example of an event handling a closure on Contact 25 (contact numbers start at zero):

```
10 REM setup event subroutine for when contact 25 closes
15 ONEVENT @CONTACT(24),GOSUB 100
20 REM do whatever here
25 GOTO 20
100 REM contact 25 closed event
105 PRINT "CONTACT(25) closed"
110 RETURN
```

This would print "CONTACT(25) closed" whenever Contact 25 closes.

In order to handle an event synchronously a program may wait for an event to occur by using the **WAIT** statement. Program execution stalls at that statement until the specified event happens. Alternatively, the program may poll the associated special variable's value in a loop looking for the event to have been signaled. Here's an example of polling for a closure on Contact 25:

```
10 REM poll contact(25) closures
15 IF @CONTACT(24) = 1 THEN 100
20 REM do whatever here
25 GOTO 15
100 REM contact 25 closed
105 PRINT "CONTACT(25) closed, clear it"
110 @CONTACT(24)=0
115 REM do whatever here
120 GOTO 15
```

This would print "CONTACT(25) closed, clear it" whenever Contact 25 closes. If you poll for events, you have to manually clear them in order to see the next one – ONEVENT handling does this clearing automatically.

The **SIGNAL** statement may be used in a program to force an event to happen.

It is very important to note that the **ONEVENT** handler subroutine executes in the context of the running program: it has access to all program variables. Since the event handler may be executed at any time in between any program statements care should be used when changing program variables from within an event handler as it may cause unexpected results in the execution of other program statements that may be using and depending upon the values of those same variables. Incorrect or unexpected program execution may result – code event handlers carefully.

See the **ONEVENT** statement definition below for a table showing what events may be processed and listing their relative priority.

Statements

ACS Basic program lines consist of an optional integer line number followed by one or more statements. Multiple statements on a line are allowed, separated by a colon ('**:**'). Only the first statement on a line may have a line number. A Direct mode of operation is available for some statements when they are entered without a line number and are executed immediately. Here are some sample program statements:

```
10 REM This is a comment
20 FOR I=0 TO 10:PRINT I:NEXT I
```

The statement keywords are 'tokenized' when entered to save program memory and speed program execution: IE: the keyword **GOSUB** would be tokenized to a single byte instead of five bytes. In addition, the statement line numbers are converted to a two-byte unsigned integer form to save space and facilitate program execution. Saved programs are expanded (un-tokenized)

The following statement keywords are supported:

CLEAR

Erase variables only.

CLOSE #N

Close file #***N*** (0-9) opened with **OPEN** statement.

DATA

Program mode only. Enter "inline" **DATA** statements holding values that can be accessed by **READ** and **ORDER** statements. All related **DATA** statements should be in a group of sequential lines.

DEL path

Delete files and directories on the Compact Flash card. The full ***path*** must be specified without a leading backslash. Directories must be empty to be deleted. ***Path*** may be a constant string or you can use a string variable as the ***path*** by concatenating it to such a string: DEL ""+P$.

DELAY value

Pause program execution for value * 20mSEC. While the delay is in process, no Events can occur.

DIM var(size)[, ...]

Dimension a numeric array **var**iable to hold **size** integers. Array variable elements may then be accessed using a numeric index in parenthesis that ranges from the first element of zero to the last element of size less one: A(0), A(1), , A(size - 1). If an attempt is made to access a variable as an array before it has been dimensioned a Dimension Error will result. If an attempt is made to access an array element with a negative index or an index beyond the currently defined array size an Index Out of Range Error will result. A variable may be re-dimensioned, however the current contents of the variable will be lost.

DIR [path]

Show files on the Compact Flash card. An optional ***path*** may be specified without a leading backslash. Wildcard characters ? and * may be used to match multiple files.

DIR #N, [path]

Write a list of files on the Compact Flash card to an open file #***N*** (0-9). An optional ***path*** may be specified without a leading backslash. Wildcard characters ? and * may be used to match multiple files.

EDIT line

Direct mode only. Using an ANSI terminal allows editing a line by displaying the statement, moving the cursor with the Home, Left arrow, Right arrow, End and Backspace keys. Typed characters are entered at the cursor. The Enter key accepts the changes, a double ESC key aborts the edit.

END

Program mode only. Terminate program with no message.

ERROR value

Force an error. Program execution stops and an error message is displayed.

```
10 ERROR 250
Ready
run
250 error in line 10
Ready
```

EXITFOR line

Program mode only. Exit out of a ***FOR/NEXT*** loop by popping the ***FOR*** off of the control stack and jumping to ***line***.

FINPUT #N, var[$], … , var[$]

Get value for one or more **var**iables from file #***N*** (0-9). Note that when an end of file occurs, the **var**iables will have their last value. Test the @FEOF(#N) specialvar to detect this condition. The data items in the file are separated by commas, with string values surrounded by double quotes. If the data in the file ends before all of the variables have been assigned values an Out of Data error occurs. Incorrect data formatting in the file can cause a Syntax Error to occur.

FOR var=init TO limit [STEP increment]

Program mode only. Perform a counted loop executing statements up until the matching **NEXT** statement. The maximum number of nested ***FOR/NEXT*** loops and ***GOSUB*** subroutines is currently 50.

FOPEN #N, recordlength, "path"

Opens filename ***path*** as fixed record length file #***N*** (0-9) for subsequent sequential / random access via **FREAD#**, **FWRITE#** statements. If ***recordlength*** is negative or greater than 127 it is forced to 127. The ***recordlength*** includes the trailing CR/LF character pair that terminates each record.

FREAD #N, recordnumber, var[$], var[$], … var[$]

Reads ASCII data from fixed length records on file ***#N*** opened by ***FOPEN #N*** into the list of variables. Before the data is read, the file is positioned to the desired ***recordnumber (0 <= recordnumber <= 32767)***. A negative ***recordnumber*** seeks to the end of the file. Reading past the current end of file sets the ***@FEOF(#N)*** specialvar and signals the associated event. The data items in the file are separated by commas, with string values surrounded by double quotes. If the data in the file ends before all of the variables have been assigned values an Out of Data error occurs. Incorrect data formatting in the file can cause a Syntax Error to occur.

FWRITE #N, recordnumber, var[$], var[$], ... var[$]

Writes ASCII data into fixed length records on file ***#N*** opened by ***FOPEN #N*** from the list of variables. Before the data is written, the file is positioned to the desired ***recordnumber (0 <= recordnumber <= 32767)***. A negative ***recordnumber*** seeks to the end of the file. The data items written to the file are separated by commas, with string values surrounded by double quotes. The record is padded with spaces to ***recordlength*** including the trailing CR/LF character pair which terminates each record. The file may be viewed using the ***TYPE*** command.

FINSERT #N, recordnumber, var[$], var[$], ... var[$]

Inserts ASCII data into fixed length records on file ***#N*** opened by ***FOPEN #N*** from the list of variables using a temporary file FINSERT.TMP. Before the data is inserted, the file is positioned to the desired ***recordnumber (0 <= recordnumber <= 32767)***, and records in the file after ***recordnumber*** are shifted down. A negative ***recordnumber*** seeks to the end of the file before inserting. The data items inserted into the file are separated by commas, with string values surrounded by double quotes. The record is padded with spaces to ***recordlength*** including the trailing CR/LF character pair which terminates each record. The file may be viewed using the ***TYPE*** command.

FDELETE #N, recordnumber

Removes fixed length record ***recordnumber (0 <= recordnumber <= 32767)*** on file ***#N*** opened by ***FOPEN #N*** using a temporary file FDELETE.TMP.

GOSUB line

Program mode only. Calls a subroutine that starts at ***line*** and ends with a **RETURN** statement. A subroutine consists of a group of program statements that start at a certain ***line*** number and end in a line with a **RETURN** statement. To call the subroutine from your program use the **GOSUB** statement which transfers program execution to the specified line number and executes those program statements until it executes a **RETURN** statement. Upon execution of the **RETURN** statement, program execution continues at the statement after the **GOSUB**. The maximum number of nested ***FOR/NEXT*** loops and ***GOSUB***s is currently 50.

GOTO line

Program mode only. Program execution continues by jumping to ***line***.

IF test THEN line/statement [ELSE line2/statement2]

Program mode only. Conditional execution jump. The expression ***test*** is evaluated, and if non-zero, program execution continues at ***line*** or ***statement*** is executed. If the optional **ELSE** clause is present program execution continues at ***line2*** or ***statement2*** if the ***test*** expression evaluates to zero.

INPUT var

Get value for variable from the serial port.

INPUT "prompt", var

Get value of **var**iable from the serial port with prompt. Prompt may be a constant string or you can use a string variable in the prompt by concatenating it to such a string: **INPUT** ""**+A$,B$**

INPUT #N, var

Get value for **var**iable from file #***N*** (0-9). Note that when an end of file occurs, the **var**iable will have its last value. Test the @FEOF(#N) specialvar to detect this condition.

[LET]var[$]=expr[$] *(default statement)*

Program or Direct mode. Sets **var**iable = **expr**ession (This is the default statement, so the **LET** keyword is not required). An attempt to assign a string value to a numeric variable or a numeric value to a string variable will generate a Type error. Some examples:

```
LET a0 = 240
100 Z9$ = "Test"
@TIMER(0) = 240
```

LIF test THEN statement[:statement]

Program mode only. Long IF (all statements to end of line). The expression **test** is evaluated, and if non-zero, the statements to the end of the program line are executed.

LIST [start, [end]]

Direct mode only. List program lines to the serial port. May also specify a starting and ending line number to limit the range of lines that are displayed.

LIST #N [start, [end]]

Direct mode only. List program lines to open file #***N*** (0-9). May also specify a starting and ending line number to limit the range of lines that are displayed.

LOAD path

Program or Direct mode. Load an ACS Basic program from a Compact Flash file specified by ***path***. The full ***path*** to the program file must be specified and must not start with a leading backslash. When **LOAD** is used within a program, execution continues with the first line of the newly loaded program. In this case, the user variables are not cleared. This provides a means of chaining to a new program, and passing information to it. When used in a program note that **LOAD** must be the last statement on a line.

```
load program1
Ready
list
10 PRINT "Program 1 A=",a
20 a=a+1
30 LOAD program2
Ready
load program2
Ready
list
10 PRINT "Program 2 A=",a:a=a+1:LOAD program1
Ready
run
Program 2 A= 0
Program 1 A= 1
Program 2 A= 2
Program 1 A= 3
 ESC at line 30
Ready
```

MD path

Direct mode only, requires a CF card. Makes a new directory on the Compact Flash card. ***Path*** must be a complete path for the new directory without the leading backslash, and it must not already exist. ***Path*** may be a constant string or you can use a string variable as the ***path*** by concatenating it to such a string: ***MD*** ""***+P$***.

MEMORY

Displays the currently available program memory and CF card memory if a CF card is present.

NEW

Direct mode only. Erase all program statements and clear variable values.

NEXT [var]

Program mode only. End of a counted loop. Statement execution resumes with the matching **FOR** statement if the step increment of the control variable has not reached the limit. Execution of a **NEXT** statement without a preceding **FOR** causes a nesting error.

ON expr, GOSUB line0, line1, line2, … ,lineN

Program mode only. Case statement dispatching via subroutines. The value of **expr** is evaluated, and a subroutine call is performed to the **line0** statement if zero, **line1** if one, etc.. If the value of **expr** is negative or greater than the number of line numbers present, execution continues with the next statement. Upon return from the GOSUB execution continues with the next statement.

```
5 REM ONGOSUB Demo
10 a=0
20 ON a,GOSUB 100,200,300,400,500
30 GOTO 20
100 PRINT "1",
105 a=a+1
110 RETURN
200 PRINT "2",
205 a=a+1
210 RETURN
300 PRINT "3",
305 a=a+1
310 RETURN
400 PRINT "4",
405 a=a+1
410 RETURN
500 PRINT "5"
505 a=0
510 RETURN
Ready
run
12345
12345
12345
12345
12345
1 ESC at line 105
Ready
```

ON expr, GOTO line0, line1, line2, … , lineN

Program mode only. Case statement dispatching via jumps. The value of **expr** is evaluated, and a jump is performed to the **line0** statement if zero, **line1** if one, etc.. If the value of **expr** is negative or greater than the number of line numbers present, execution continues with the next statement.

```
5 REM ON GOTO DEMO
10 a=0
20 ON a,GOTO 100,200,300,400,500
30 GOTO 10
100 PRINT "1",
105 a=a+1
110 GOTO 20
200 PRINT "2",
205 a=a+1
210 GOTO 20
300 PRINT "3",
305 a=a+1
310 GOTO 20
400 PRINT "4",
405 a=a+1
410 GOTO 20
500 PRINT "5"
505 a=a+1
510 GOTO 20
Ready

run
12345
12345
12345
12345
1234 ESC at line 20
Ready
```

ONERROR GOTO line

Program mode only. Provides one-shot error handling. Upon any error, statement execution starts at line, and the ERR() function has the value of the error number and the ERR$() function has the string version of the error number. The **ONERROR** condition is then cleared so that subsequent errors result in program termination.

```
10 ONERROR GOTO 100
20 REM error follows
30 a=10/0
40 STOP
100 PRINT "Error #",ERR()," - ",ERR$()
Ready
run
Error # 6 - Divide by zero error in line 30
Ready
```

ONEVENT @specialvar, GOSUB line

Program mode only. Provides semi-asynchronous event handling via subroutines. Certain ACS Basic special variables can trigger events. The **ONEVENT** statement allows the event to be associated with the execution of a subroutine. When the event occurs, after execution of any current statement that does not transfer control, control is transferred to the subroutine starting at ***line***. *While in the event subroutine, only higher priority events will be recognized until after the **RETURN** statement is executed.* An event handler can be disabled by specifying a ***line*** number of zero.

The following special variables can cause events and are listed in order of decreasing priority:

@SOUNDFRAMESYNC	event occurs every @SOUNDFRAMEPRESCALER x 20mSEC while a sound is playing
@TIMER(x)	event occurs one time whenever the timer counts down to zero. Special variable **@TIMER(0)** is the highest priority, followed by **@TIMER(1)**, … then **@TIMER(9)**. 0 <= x <= 9
@CLOSURE(x)	event occurs whenever the associated CFSound-3 contact has closed. 0 <= x <= 55
@OPENING(x)	event occurs whenever the associated CFSound-3 contact has opened. 0 <= x <= 55
@FEOF(#N)	Event occurs after INPUT #N, FINPUT #N or FREAD #N reaches the end of file #N
@SECOND	event occurs once per second.
@MINUTE	event occurs once per minute.
@HOUR	event occurs once per hour.
@DOW	event occurs once per day at midnight.
@DATE	event occurs once per day at midnight.
@MONTH	event occurs once per month at midnight of day 1.
@YEAR	event occurs once per year.
@SOUND$	event occurs after the last queued **@SOUND$** sound has finished playing.
@MSG$	event occurs after receipt of a serial character stream delineated by the **@SOM** and **@EOM** characters.

Here is a short program that outputs the current time, once per second, on the serial port. Note that the program's idle loop, which it executes while waiting for the second event to occur, consists of a single GOTO self statement.:

```
5 REM print the time once per second
10 ONEVENT @SECOND,GOSUB 100
20 GOTO 20
100 PRINT CHR$(13),
105 PRINT FMT$("%2d",@HOUR),
110 PRINT ":",
115 PRINT FMT$("%02d",@MINUTE),
120 PRINT ":",
125 PRINT FMT$("%02d",@SECOND),
130 RETURN
Ready
run
14:47:15 ESC at line 30
Ready
```

OPEN #N, "path", "options"

Open filename ***path*** as file #***N*** (0-9) for subsequent access via **DIR #N**, **INPUT #N, FINPUT #N** or **PRINT #N** statements. The ***options*** string characters are:

"r"	opens file for reading, if ***path*** does not exist an error is generated
"w"	opens file for writing, if ***path*** exists its contents are destroyed
"r+"	opens file for read and write, the ***path*** must exist
"w+"	opens an empty file for read and write, if ***path*** exists its contents are destroyed
"a+"	opens file for reading and appending (seek to end of file after open)
"b"	opens file in binary mode, no translations
"t"	opens file in text mode (default), CR/LF pairs are translated to LF on input and LF translated to CR/LF pairs on output.

ORDER line

Program mode only. This statement positions the read data pointer to statement ***line*** number. The statement at ***line*** must be a series of one or more **DATA** statement.

PLAY file

Plays the sound ***file*** and waits until it completes. Program execution then continues with the next statement. If the ***file*** is not a valid .WAV file of the correct format, sample rate and sample size for the CFSound-3 a Sound File error is generated. ***File*** may be a constant string or you can use a string variable as the ***file*** by concatenating it to such a string: ***PLAY*** ""**+*P$***.

PRINT expr[, expr ...]

Prints one or more expression(s) to the serial port. If the statement ends in a comma (",") no Carriage Return / Line Feed pair is appended to the printed expressions allowing multiple print statements to display on the same line.

PRINT#N, expr[, expr ...]

Prints one or more expressions to a previously opened file #***N*** (0-9).

READ var[,var ...]

Program mode only. Reads data from program statements into **var**iables. You MUST issue an **ORDER** statement targeting a line containing a valid **DATA** statement before using **READ**.

RETURN

Program mode only. Return from a subroutine invoked via a **GOSUB** statement. A return without a prior **GOSUB** will generate a Stack error.

REM

Comment... the remainder of line is ignored. Used to document the operation of the program.

REN oldfile, newfile

Renames oldfile to newfile. ***Oldfile*** and ***newfile*** may be constant strings or you can use string variables as the ***files*** by concatenating them to empty strings: **REN** ""**+O$,** ""**+N$**.

RUN [line]

Direct mode only. Executes the program starting at the lowest or optional ***line*** number.

SAVE [path]

Direct mode only. Saves the current program to a disk file on the Compact Flash card with the filename specified in ***path***.

SIGNAL @specialvar

Signal an event associated with Special variable.

STOP

Program mode only. Terminates the program and issues a message.

```
10 a=a+1
20 STOP
Ready
run
STOP in line 20
Ready
```

TYPE path

Displays the contents of a CF card filename named ***path*** as ASCII characters on the serial port. ***Path*** may be a constant string or you can use a string variable as the ***path*** by concatenating it to such a string: ***TYPE*** ""***+P$***. A double escape sequence will stop the file display.

WAIT @specialvar

Execution pauses at this statement until the associated Special variable has been signaled. In this example, program execution would pause at line 110 until all of the queued sounds had finished playing:

```
10 @SOUND$="one.wav"
20 @SOUND$="two.wav"
30 @SOUND$="three.wav"
40 @SOUND$="four.wav"
50 @SOUND$="five.wav"
60 @SOUND$="six.wav"
70 @SOUND$="seven.wav"
80 @SOUND$="eight.wav"
90 @SOUND$="nine.wav"
100 @SOUND$="ten.wav"
110 WAIT @SOUND$
```

LCDx Statements

The following LCD commands operate on an ACS LCD display connected to the CFSound-3 serial port that is configured for SOH/ETX protocol. The commands generate and send formatted strings of ASCII characters that the connected LCD display interprets to perform the operation. The proper command formatting for the attached display is controlled by the current value of the **@LCDTYPE** special variable. The **@LCDADDRESS** special variable may also optionally be used to selectively address multiple displays by inserting the display address into the generated commands. See the ACS-LCD-128x64 or ACS-LCD-320x240 Display User Manuals for additional information about these command's arguments.

LCDPRINT row[s], col, font, type, justify, expr (@LCDTYPE=0)

LCDPRINT rowstart, col, font, type, justify, expr (@LCDTYPE=1)

Displays an **expr** on an ACS LCD display connected to the CFSound-3 serial port.

LCDUNPRINT row[s], col, font, type, justify, expr (@LCDTYPE=0)

LCDUNPRINT rowstart, col, font, type, justify, expr (@LCDTYPE=1)

Un-displays an **expr** on an ACS LCD display connected to the CFSound-3 serial port.

LCDCLEAR row[s], colstart, colend (@LCDTYPE=0)

LCDCLEAR rowstart, rowend, colstart, colend (@LCDTYPE=1)

Clears an area of the screen on an an ACS LCD display connected to the CFSound-3 serial port.

LCDGRAPHIC row[s], col, data (@LCDTYPE=0 only)

Displays a byte of data on an ACS LCD display connected to the CFSound-3 serial port. This command is not supported on the ACS LCD-320x240 display.

LCDLINE startx, starty, endx, endy, color

Displays a line on an ACS LCD display connected to the CFSound-3 serial port.

LCDBOX corner1x, corner1y, corner2x, corner2y, color

Displays a box on an an ACS LCD display connected to the CFSound-3 serial port.

LCDPIXEL x, y, color

Displays a pixel on an ACS LCD display connected to the CFSound-3 serial port..

LCDCIRCLE x, y, radius, color

Draws a circle on an ACS LCD display connected to the CFSound-3 serial port.

LCDTONE frequency, duration

Produces a tone on an ACS LCD display connected to the CFSound-3 serial port.

LCDSAVE page

Saves a screen on an ACS LCD display connected to the CFSound-3 serial port.

LCDRESTORE page

Restores a screen on an ACS LCD display connected to the CFSound-3 serial port.

LCDBITMAP startrow, col, "path"

Displays a Windows .BMP bitmap file named ***path*** starting at ***startrow***(0 – 7) and ***col***umn on an ACS LCD display connected to the CFSound-3 serial port. Only mono, 16-color and 256 color bitmaps are supported. Any pixel whose color is not R=255, G=255, B=255 (white) will be displayed as an on pixel (black). Issues multiple LCD display Horizontal Load commands to image the bitmap on the display.

Operators

ACS Basic supports the following operators listed in priority from highest to lowest. Operators are evaluated in order of priority with higher priority operators executed before lower priority operators. Parenthesis may be used to change or enforce execution priority:

Operator	Description	Priority
NOT	Logical NOT	7
-	Unary minus (negate)	7
~	Bitwise NOT (1 s complement)	7
*** / %**	Multiplication, division, modulus	6
+	Addition, string concatenation	5
-	Subtraction	5
<< >>	Left Shift, Right Shift	4
= <>	Assign / test equal, test NOT equal (numeric or string)	3
< <= > >=	LT, LE, GT, GE (numeric or string)	3
& | ^	Bitwise AND, OR, Exclusive OR	2
AND OR	Logical AND, OR	1

The 'test' relational operators (=, <>, <, <=, >, >=) can be used in any expression, and evaluate to 1 if the test condition is TRUE, and 0 if it is FALSE. The IF and LIF commands accept any non-zero value to indicate a TRUE condition. Note that multiple 'test' operators can be combined with the logical AND, OR operators and suitable parenthesis.

Expressions

In ACS Basic expressions consist of one or more variables, constants, functions or special variables that may optionally be joined together by Operators. The evaluation order may be controlled by the judicious use of parenthesis. Expressions may be nested up to 10 levels. Some examples:

```
a=10
Ready
print a*30
 300
Ready
print fmt$("%02X", a)
0A
Ready
print a<<2
 40
Ready
print (a<<2)=0
 0
Ready
print (a<<2)<>0
 1
Ready
print a^4
 14
Ready
```

Functions

ACS Basic provides several functions that may be used in expressions. There must not be a space between the function name and the opening parenthesis. Functions must be used in a statement such as a LET or PRINT they cannot be executed standalone in immediate mode.

ASC(char)

Returns the numeric ASCII value of the character argument.

ABS(expr)

Returns the absolute value of the numeric argument.

CHR$(expr)

Returns an ASCII string containing the character equivalent of the expression argument.

ERR( )

Returns the last error number.

ERR$( )

Returns the string representation of the last error number.

FMT$(fmt$, expr[$])

Returns a formatted ASCII string of **expr**ession using format specification **fmt$**. A format specification consists of [optional] and required fields and has the following form:

% [Flags] [Width] [.Precision] Type

Each field of a format specification is a single character or a number signifying a particular format option. The simplest format specification contains only the percent sign and a type character (for example, %d). If a percent sign is followed by a character that has no meaning as a format field, the character is copied to the return value. For example, to produce a percent sign in the return value, use %%.

The optional fields, which appear before the *type* character, control other aspects of the formatting, as follows:

<table>
<tr><td>Type</td><td>Required character that determines whether the associated argument is interpreted as a character, a string, or a number:
<table>
<tr><td>c</td><td>character</td></tr>
<tr><td>d</td><td>signed decimal integer</td></tr>
<tr><td>i</td><td>signed decimal integer</td></tr>
<tr><td>u</td><td>unsigned decimal integer</td></tr>
<tr><td>s</td><td>string</td></tr>
<tr><td>o</td><td>unsigned octal integer</td></tr>
<tr><td>x</td><td>unsigned hexadecimal integer</td></tr>
<tr><td>X</td><td>unsigned HEXADECIMAL integer</td></tr>
</table></td></tr>
<tr><td>Flags</td><td>Optional character or characters that control justification of output and printing of signs, blanks, and octal and hexadecimal prefixes. More than one flag can appear in a format specification.
<table>
<tr><td>-</td><td>left align the result in the given field width</td></tr>
<tr><td>+</td><td>prefix the output with a sign (+/-) if the type is signed</td></tr>
<tr><td>0</td><td>if Width is prefixed with 0, zeros are added until the minimum width is reached. If 0 and – appear, the 0 is ignored. If 0 is specified with an integer format, the 0 is ignored.</td></tr>
<tr><td>blank(‘ ’)</td><td>prefix the output with a blank if the result is signed and positive; the blank is ignored if both the blank and + flags appear</td></tr>
<tr><td>#</td><td>when used with o, x or X format, prefix any nonzero output value with 0, 0x or 0X respectively, otherwise ignored</td></tr>
</table></td></tr>
<tr><td>Width</td><td>Nonnegative decimal integer controlling the minimum number of characters printed. If the number of characters in the output value is less than the specified width, blanks are added to the left or the right of the values — depending on whether the – flag (for left alignment) is specified — until the minimum width is reached. If Width is prefixed with 0, zeros are added until the minimum width is reached (not useful for left-aligned numbers). The Width specification never causes a value to be truncated. If the number of characters in the output value is greater than the specified width, or if Width is not given, all characters of the value are printed (subject to the Precision specification).</td></tr>
<tr><td>Precision</td><td>Specifies a nonnegative decimal integer, preceded by a period (.), which specifies the number of characters to be printed, the number of decimal places, or the number of significant digits. Unlike the Width specification, the precision specification can cause truncation of the output value. If Precision is specified as 0 and the value to be converted is 0, the result is no characters output.
<table>
<tr><td>c</td><td>Precision has no effect</td></tr>
<tr><td>d,i,u,o, x,X</td><td>Precision specifies the minimum number of digits to be output. If the number of digits is less than Precision, the output is padded on the left with zeroes. The value is not truncated when the number of digits exceeds Precision</td></tr>
<tr><td>s</td><td>Precision specifies the maximum number of characters to be output. Characters in excess of Precision are not output</td></tr>
</table></td></tr>
</table>

MID$(var$, start, len)

Returns a string consisting of **len**gth number of characters of string **var**iable from zero based ***start*** character position.

LEFT$(var$, len)

Returns a string containing the leftmost **len**gth characters of string **var**iable.

RIGHT$(var$, len)

Returns a string containing the rightmost **len**gth characters of string **var**iable.

INSERT$(var$, start, var2$)

Returns a string ***var***iable with the contents of ***var***iable***2*** inserted at zero based position ***start***.

```
10 REM test insert$
20 s$ ="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
30 i$ ="insert"
35 REM insert at beginning
40 PRINT INSERT$(s$,0,i$)
45 REM insert in middle
50 PRINT INSERT$(s$,13,i$)
55 REM insert past end
60 PRINT INSERT$(s$,30,i$)
Ready
run
insertABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMinsertNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZinsert
Ready
```

REPLACE$(var$, start, var2$)

Returns a string ***var***iable with the contents of ***var***iable***2*** overwritten at zero based position ***start***.

```
10 REM test replace$
20 s$ ="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
30 r$ ="replace"
35 REM replace at beginning
40 PRINT REPLACE$(s$,0,r$)
45 REM replace in middle
50 PRINT REPLACE$(s$,13,r$)
55 REM replace past end
60 PRINT REPLACE$(s$,30,r$)
Ready
run
replaceHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMreplaceUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZreplace
Ready
```

LEN(var$)

Returns the length (number of characters) of string **var**iable.

FIND(var$, searchvar$)

Returns the zero based position of string **searchvar**iable in string **var**iable or -1 if the searchvariable was not found.

RND(expr)

Returns a psuedo random number that ranges from 0 to (**expr**ession - 1).

VAL(expr$)

Returns the numeric value of the string argument representation of a number.

STR$(expr)

Returns a string representation of the numeric argument.

MULDIV(number, multiplier, divisor)

Returns a 16 bit result of ((number * multiplier) / divisor) where number, multiplier and divisor are 32-bit internally. Useful for calculating percentages, etc., where the normal multiply would overflow a signed 16-bit number.

```
10 REM calculate 55 percent of 999
20 PRINT MULDIV(999,55,100),".",MULMOD(999,55,100)
Ready
run
 549. 45
```

MULMOD(number, multiplier, divisor)

Returns a 16 bit result of ((number * multiplier) % divisor) where number, multiplier and divisor are 32-bit internally. Useful for calculating remainders of percentages, etc., where the normal multiply would overflow a signed 16-bit number.

Errors

The following errors can be produced. The placeholder 'dd' in the message is replaced with the line number where the error was detected if the error was encountered in a running program. Some Syntax Errors will provide additional information after the line number further identifying the error:

Error #	Error Message	Causes
1	"Syntax error in line dd"	Incorrect statement format
2	"Illegal program command error in line dd"	Direct mode only statement in program mode
3	"Illegal direct command error in line dd"	Program mode only statement in direct mode
4	"Line number error in line dd"	Target line number not in program
5	"Wrong expression type error in line dd"	Numeric value when String expected or vice versa
6	"Divide by zero error in line dd"	Division by zero
7	"Nesting error in line dd "	NEXT without preceding FOR, RETURN without preceding GOSUB
8	"File not open error in line dd "	CLOSE#, LIST#, PRINT# or INPUT# without successful OPEN statement
9	"File already open error in line dd "	OPEN# on already open file
10	"File # Out of Range in line dd"	File # out of range 0 - 9
11	"Input error in line dd "	Numeric value expected in INPUT # statement
12	"Dimension error in line dd "	Dimension error
13	"Index out of range in line dd"	Subscript out of range
14	"Data error in line dd "	ORDER line # not DATA statement, READ past DATA statements
15	"Out of memory error in line dd "	Insufficient memory
16	"No File System error in line dd "	ACS Basic running without CF card
17	"Unknown @var error in line dd "	Unknown special variable
18	"Timer # out of range error in line dd "	@TIMER(x) subscript out of range 0 - 9
19	"Port # out of range error in line dd "	@PORT(x) subscript out of range 0 - 255
20	"Contact # out of range error in line dd "	@CONTACT(x), @CLOSURE(x), @OPENING(x) subscript out of range
21	"Stack Overflow error in line dd "	Too many nested FOR and/or GOSUB and/or events
22	"No CF card error in line dd "	Statement requiring Compact Flash card with no card detected
23	"Invalid .WAV file error in line dd "	.WAV file format not 44.1KHz 16-bit mono or stereo
24	"LCDx arguments Out of Range error in line dd"	One or more argument to a LCDx statement are out of range
25	"FWRITE record # Out of Range error in line dd"	Attempt to FWRITE to a record number that is past the immediate end of file
26	"FWRITE exceeds record length error in line dd"	Length of data in FWRITE variables list including commas and quotes exceeds the recordlength specified in the associated FOPEN
27	"FINSERT record # Out of Range error in line dd"	Attempt to FINSERT to a record number that is past the immediate end of file
28	"FINSERT exceeds record length error in line dd"	Length of data in FINSERT variables list including commas and quotes exceeds the recordlength specified in the associated FOPEN
29	"FDELETE past end of file error in line dd "	FDELETE record number exceeds file length
30	"Can't delete file error in line dd"	Can't delete file
31	"Can't make directory error in line dd"	Can't create directory
32	"Can't rename file error in line dd"	Can't rename file
33	"No DMX module error in line dd"	@DMX--- specialvar access attempted with no DMX I/O module present
34	"DMX Channel # Out of Range error in line dd"	@DMXDATA(x) access where x >= 511
35	"DMX Analog # Out of Range error in line dd"	@DMXANALOG(x) access where x >= 7
36	"DMX Analog # Read Only error in line dd"	Attempt to set @DMXANALOG(x)
37	"Unknown Command error in line dd"	ACS Basic doesn't recognize the command
38	"Can't use @VAR in line dd"	Illegal use of specialvar in FOR, DIM, INPUT, READ, FREAD or FINPUT statement
39	"Mis-matched quotes in line dd"	Missing one of a pair of double quotes delimiting a string
40 - 65535	"x error in line dd"	ERROR x statement

Examples

Here are a few sample programs that illustrate the various language features and what can be done with some simple lines of code.

Setting the Real Time Clock

Set the CFSound-3's Real-Time-Clock with this short program. The program prompts for the values of the Month, Date, Year, Hour, Minute and Second while range checking the values, then displays the formatted time on the connected ANSI terminal once a second.

```
5 REM set the cfsound rtc
7 INPUT "set the RTC first (y/n):", s$
8 IF s$="y" THEN 20
9 IF s$="Y" THEN 20
10 GOTO 110
20 INPUT "month (1-12):",m
25 IF m <1 THEN 20
27 IF m >12 THEN 20
30 @MONTH=m
35 INPUT "date (1-31):",d
40 IF d <1 THEN 35
42 IF d >31 THEN 35
45 @DATE=d
50 INPUT "year (00-99):", y
52 IF y <0 THEN 50
53 IF y >99 THEN 50
60 @YEAR=y
65 INPUT "hour (00-23):",h
70 IF h <0 THEN 65
72 IF h >23 THEN 65
75 @HOUR=h
80 INPUT "minute (00-59):",m
85 IF m <0 THEN 80
87 IF m >59 THEN 80
90 @MINUTE=m
95 INPUT "second (00-59):",s
100 IF s <0 THEN 95
102 IF s >59 THEN 95
105 @SECOND=s
110 ONEVENT @SECOND,GOSUB 1000
115 a=0
120 GOTO 115
1000 PRINT CHR$(13),
1002 ON @DOW,GOSUB 2000,2001,2002,2003,2004,2005,2006
1005 ON @MONTH,GOSUB 1200,1201,1202,1203,1204,1205,1206,1207,1208,1209,1210,1211,1212
1010 PRINT d$+" "+m$+FMT$(" %2d",@DATE)+FMT$(", %02d",@YEAR),
1015 PRINT FMT$(" %2d", @HOUR)+":"+FMT$("%02d",@MINUTE)+":"+FMT$("%02d",@SECOND),
1020 RETURN
1200 m$="???":RETURN
1201 m$="JAN":RETURN
1202 m$="FEB":RETURN
1203 m$="MAR":RETURN
1204 m$="APR":RETURN
1205 m$="MAY":RETURN
1206 m$="JUN":RETURN
1207 m$="JUL":RETURN
1208 m$="AUG":RETURN
1209 m$="SEP":RETURN
1210 m$="OCT":RETURN
1211 m$="NOV":RETURN
1212 m$="DEC":RETURN
2000 d$="SUN":RETURN
2001 d$="MON":RETURN
2002 d$="TUE":RETURN
2003 d$="WED":RETURN
2004 d$="THU":RETURN
2005 d$="FRI":RETURN
2006 d$="SAT":RETURN
```

Two Sound Sequences

The CFSound-3 can play a single sequence of sounds in CFSound Mode using a CFSOUND.INI file to configure the sequence contact number and sound range. Here's a simple ACS Basic program that will allow two different sequences each controlled by a built-in contact.

Remember that the @CLOSURE(x) special variable index argument x is zero based, so for Contact #25 the x value would be 24, etc..

Contact #25 activations cycle through sounds ONE.WAV, TWO.WAV, THREE.WAV and FOUR.WAV, and contact #26 activations cycle through sounds FIVE.WAV, SIX.WAV, SEVEN.WAV and EIGHT.WAV.

Here's how it works. Lines 10 and 20 setup event handlers for contact closures on contacts #25 and #26. The subroutine at line 1000 is called whenever a closure is detected on contact #25, the subroutine at line 2000 is called whenever a closure is detected on contact #26. Line 30 clears the two sequence variables that keep track of what sound to play next. The variable S0 keeps track of what sound to play for contact #25, and S1 tracks the sounds for contact #26. When a closure is detected on contact #25, the subroutine at line 1000 stops any currently playing sound by clearing the @SOUND$ special variable. Line 1010 then starts playing the next sound in the sequence based upon the current value of S0, and advances the value of S0 for the next contact closure. When a closure is detected on contact #26, the subroutine at line 2000 stops any currently playing sound by clearing the @SOUND$ special variable. Line 2010 then starts playing the next sound in the sequence based upon the current value of S1, and advances the value of S1 for the next contact closure.

```
5 REM play two sequences off of the two built-in rear contacts
10 ONEVENT @CLOSURE(24), GOSUB 1000
20 ONEVENT @CLOSURE(25), GOSUB 2000
30 S0 = 0: S1 = 0
40 GOTO 40
1000 REM contact #25's sequence
1005 @SOUND$=""
1010 ON S0,GOSUB 1100,1105,1110,1115
1015 S0 = S0 + 1
1020 IF S0 > 3 THEN S0=0
1025 RETURN
1100 @SOUND$="ONE.WAV" : RETURN
1105 @SOUND$="TWO.WAV" : RETURN
1110 @SOUND$="THREE.WAV" : RETURN
1115 @SOUND$="FOUR.WAV" : RETURN
2000 REM contact #26's sequence
2005 @SOUND$=""
2010 ON S1,GOSUB 2100,2105,2110,2115
2015 S1 = S1 + 1
2020 IF S1 > 3 THEN S1=0
2025 RETURN
2100 @SOUND$="FIVE.WAV" : RETURN
2105 @SOUND$="SIX.WAV" : RETURN
2110 @SOUND$="SEVEN.WAV" : RETURN
2115 @SOUND$="EIGHT.WAV" : RETURN
```

Activating Multiple Output Contacts for a Sound

The CFSound-3 can activate a single output contact when a sound is played in CFSound mode. Here's a simple ACS Basic program that will allow multiple output contacts to be controlled when a sound plays.

Remember that the @CLOSURE(x) special variable index argument x is zero based, so for Contact #25 the x value would be 24, etc.. This example assumes that the CFSound-III is equipped with a Contact I/O 8 module installed on the rear expansion connector to provide output contacts 0 – 7.

A closure on contact #25 plays sound ONE.WAV and activates output contacts 1 and 2 while the sound is playing. A closure on contact #26 plays sound TWO.WAV and activates output contacts 1 and 3 while the sound is playing.

Here's how it works. The program runs a loop in lines 10 through 30 looking to see if an input closure was detected on contacts #25 and #26. A closure on contact #25 jumps to line 100. A closure on contact #26 jumps to line 200. This process is referred to as 'polling' the input contacts for closures. Starting at line 100 the desired output contacts are activated, then the sound is played, then the output contacts are deactivated. The contact closure is cleared, and the program starts polling again. The same process is programmed starting at line 200 for the other contact and desired output contact configuration.

```
5 REM Poll the two contact inputs for closures
10 IF @CLOSURE(24) THEN GOTO 100
20 IF @CLOSURE(25) THEN GOTO 200
30 GOTO 10
100 REM Input 25 had a closure
110 @CONTACT(0)=1:@CONTACT(1)=1
120 PLAY "ONE.WAV"
130 @CONTACT(0)=0:@CONTACT(1)=0
140 @CLOSURE(24)=0
150 GOTO 10
200 REM Input 26 had a closure
210 @CONTACT(0)=1:@CONTACT(2)=1
220 PLAY "TWO.WAV"
230 @CONTACT(1)=0:@CONTACT(2)=0
240 @CLOSURE(25)=0
250 GOTO 10
```

Control from a Serial Port

The CFSound-3 can be controlled by serial commands in CFSound mode. If your application requires custom functionality in addition to being controlled by serial commands use the @MSG$ special variable to implement a serial protocol. This example shows a simple three character serial protocol that is used to play specific sounds and activate the push to talk relay while the sounds are playing.

The protocol consists of a single character sound number delimited by the default @SOM and @EOM characters. This yields a message structure of an ASCII Start of Header (SOH) character (CTRL-A), followed by the ASCII number of the sound to play ('1' – '4'), followed by a ASCII End of Text (ETX) character (CTRL-C). The files "ONE.WAV", "TWO.WAV", … , "FOUR.WAV" are on the CF card.

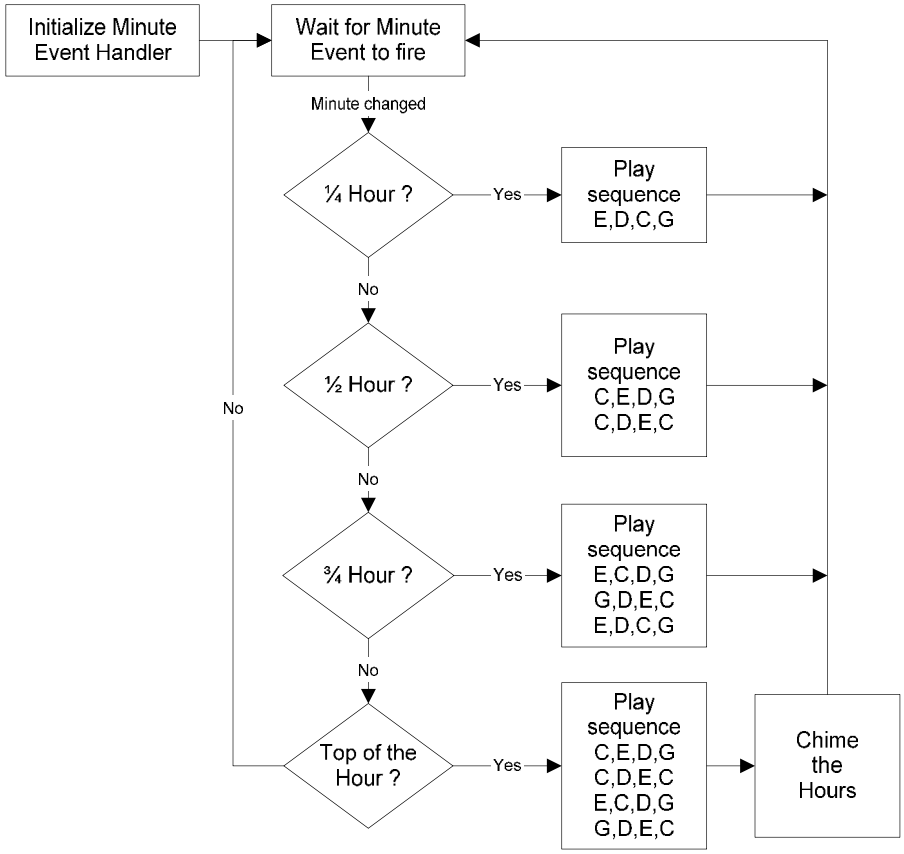
Here's how it works. An event handler is setup in line 20 – when a character string delimited by the @SOM and @EOM characters is received, control transfers to line 50 with the @MSG$ variable holding the inner contents of the string. Line 60 copies the string and resets the @MSG$ variable for receipt of the next message. The message number is converted from a string to a number in line 70, and is adjusted so that it is zero-based. Line 80 calls the subroutine matching the numeric value – the called subroutine activates the PTT relay, plays the sound, deactivates the PTT relay and returns. Line 90 then returns from the @MSG$ event handler.

```
10 REM setup @MSG$ event handler
20 ONEVENT @MSG$,GOSUB 50
30 GOTO 30
50 REM @MSG$ event handler
60 n$=@MSG$:@MSG$=""
70 n=VAL(N$)-1
80 ON n,GOSUB 100,200,300,400
90 RETURN
100 @PTT=1:PLAY "ONE.WAV":@PTT=0:RETURN
200 @PTT=1:PLAY "TWO.WAV":@PTT=0:RETURN
300 @PTT=1:PLAY "THREE.WAV":@PTT=0:RETURN
400 @PTT=1:PLAY "FOUR.WAV":@PTT=0:RETURN
```

Westminster Chimes

Turn the CFSound-3 into a digital audible clock with this short program. The program plays a chime melody using pre-recorded waveforms to emulate the Big Ben clock in London. It plays a portion of the Westminster chimes on the quarter hour, and the entire melody at the top of the hour along with chiming the hour.

Here is a flowchart of the program's logic:



Looking at the diagram, you can see that you need five different note sequences, and the Hours chime. The note sequences can be generated using individual wave files for each note, or recorded or synthesized as short sequences. In this example, Cool Edit Pro was used to capture a bell sound, shorten its envelope, then generate the musical note sequences and the hours chime sound. The five sequence sound files and hours chime are named:

```
dir *.wav
SEQ_GDEC.WAV     581954 A      08-23-2006 16:45:44
SEQ_CDEC.WAV     581954 A      08-23-2006 16:43:50
SEQ_CEDG.WAV     581954 A      08-29-2006 10:17:18
SEQ_ECDG.WAV     581954 A      08-23-2006 16:44:54
SEQ_EDCG.WAV     581954 A      08-23-2006 16:42:58
HOURS.WAV        264434 A      08-23-2006 16:42:24
-----------------------
                    6 files
                    0 directories
```

Here's how it works. The Acs Basic program initializes a line number of an event handler for the @MINUTE special variable that will be fired whenever the @MINUTE changes. It then falls into a loop waiting for the event to fire. Other statements can be executed while waiting, but to keep this example simple, it doesn't do anything else while waiting.

```
5 REM setup event handler
10 ONEVENT @MINUTE,GOSUB 100
```

```
15 REM wait here for event
20 GOTO 15
```

Whenever the @MINUTE changes, the program performs a GOSUB to the event handler program line. The event handler calculates the period of the hour by dividing the current minutes value by 15 minutes per period, and the minutes remaining in the period (remainder) by taking the modulo of the current minutes by 15. If the remainder is zero, then it is the start of a new period, and the event handler branches to the line number for the current period. If the remainder is not zero, the event handler returns. Note that the four decision diamonds above are collapsed into the single program line 110:

```
100 REM calculate period and remainder
102 p=(@MINUTE/15):r=(@MINUTE%15)
105 REM if remainder=0 then branch on period #
110 IF r=0 THEN ON p,GOTO 200,300,400,500
120 RETURN
```

For the quarter past, half past and three quarter past periods, the handler queues the appropriate note sequences to be played and returns. For the top of the hour, the handler queues the note sequences, and then queues the chime sound a number of times to match the hour. It then returns:

```
200 REM play whole sequence & chime hour
202 @SOUND$="SEQ_CEDG.WAV"
204 @SOUND$="SEQ_CDEC.WAV"
206 @SOUND$="SEQ_ECDG.WAV"
208 @SOUND$="SEQ_GDEC.WAV"
210 h=@HOUR:IF h>12 THEN h=h-12
211 IF h=0 THEN h=12
212 FOR c=h TO 1 STEP -1
215 @SOUND$="HOURS.WAV"
220 NEXT c
225 RETURN
300 REM play quarter past sequence
305 @SOUND$="SEQ_EDCG.WAV"
310 RETURN
400 REM play half past sequence
402 @SOUND$="SEQ_CEDG.WAV"
405 @SOUND$="SEQ_CDEC.WAV"
410 RETURN
500 REM play three quarters past sequence
502 @SOUND$="SEQ_ECDG.WAV"
504 @SOUND$="SEQ_GDEC.WAV"
506 @SOUND$="SEQ_EDCG.WAV"
510 RETURN
```

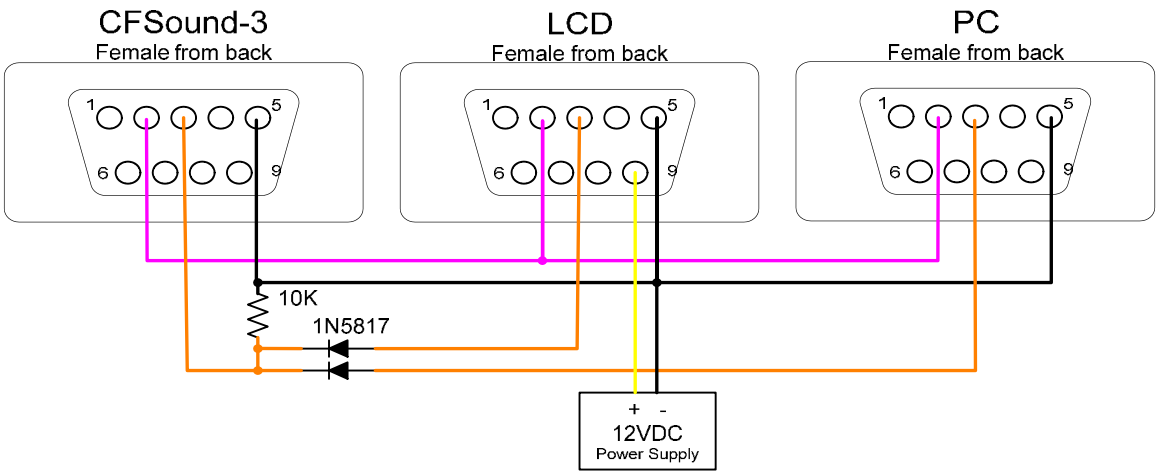
Renaming the program to CFSOUND.BAS and placing it along with the requisite sound files onto the CF card will turn your CFSound-3 into a Big Ben clock. Here's the entire program:

```
5 REM setup event handler
10 ONEVENT @MINUTE,GOSUB 100
15 REM wait here for event
20 a=0:GOTO 15
100 REM calculate period and remainder
102 p=(@MINUTE/15):r=(@MINUTE%15)
105 REM if remainder=0 then branch on period #
110 IF r=0 THEN ON p,GOTO 200,300,400,500
120 RETURN
200 REM play whole sequence & chime hour
202 @SOUND$="SEQ_CEDG.WAV"
204 @SOUND$="SEQ_CDEC.WAV"
206 @SOUND$="SEQ_ECDG.WAV"
208 @SOUND$="SEQ_GDEC.WAV"
210 h=@HOUR:IF h>12 THEN h=h-12
211 IF h=0 THEN h=12
212 FOR c=h TO 1 STEP -1
215 @SOUND$="HOURS.WAV"
220 NEXT c
225 RETURN
300 REM play quarter past sequence
305 @SOUND$="SEQ_EDCG.WAV"
310 RETURN
400 REM play half past sequence
402 @SOUND$="SEQ_CEDG.WAV"
```

```
405 @SOUND$="SEQ_CDEC.WAV"
410 RETURN
500 REM play three quarters past sequence
502 @SOUND$="SEQ_ECDG.WAV"
504 @SOUND$="SEQ_GDEC.WAV"
506 @SOUND$="SEQ_EDCG.WAV"
510 RETURN
```

Jukebox with Display

Turn the CFSound-3 into a jukebox with display using this short program. The hardware consists of a CFSound-3 and the ACS-LCD-128x64 display with membrane switch, wired together with a serial cable. If a special cable is constructed, the PC can also be connected allowing for interactive software development. The following cable allows both the PC and the LCD to 'talk' to the CFSound-3 by using two diodes and a resistor for implement a wired-or of the LCD and PC TxD signals. This allows the PC to communicate with the CFSound-3 via Window's Hyperterminal accessory, and the LCD keystrokes to be sent to the CFSound-3 to interact with the Basic program:



The program captures a directory listing of the .WAV files present on the flash card and displays this listing on the LCD display. The Up and Down arrows on the membrane switch scroll the 'selection', shown in inverse font. Pressing the Enter key between the arrows plays the current selection. Several subroutines are used to simplify the main program logic.

Here's how it works. The program starts by clearing the LCD display and installing an event handler for the @MSG$ special variable:

```
1 REM
2 REM LCD Jukebox Demo
3 REM
10 REM clear display, install @msg$ handler
15 LCDCLEAR 255,0,127
25 ONEVENT @MSG$,GOSUB 8005
```

The ACS LCD Display frames its sent messages in a SOH / ETX character pair, which is the default value of the @SOM and @EOM special variables. When the program is running and not processing a Basic INPUT statement, characters received on the CFSound's serial port are processed looking for an @SOM / @EOM delimited message string. When such a message is detected, the @MSG$ variable receives the content of the message, and the @MSG$ event handler is signaled. This causes program execution to GOSUB to line 8005 after the current statement is finished.

Next the program generates a file that contains the directory of .WAV files present on the flash card, and then displays this list. The program then falls into an idle loop, waiting for LCD membrane switch key press messages to process:

```
30 REM generate and display wav file list
35 GOSUB 6005:GOSUB 7005
100 a=0
110 GOTO 100
```

The subroutine to generate the .WAV directory file opens a text file WAVES.TXT for destructive writing, directing the output of the DIR command into that file. The file is then re-opened for reading, and

the number of lines in the file are counted, subtracting 3 lines for the summary lines at the bottom of the DIR command:

```
6000 REM
6001 REM Generate list of .WAV files
6002 REM
6005 OPEN #0,"WAVES.TXT","w"
6010 DIR #0,*.wav
6015 CLOSE #0
6020 n=0:OPEN #0,"WAVES.TXT","r"
6025 INPUT #0,l$
6030 LIF LEN(l$) >0 THEN n=n+1:GOTO 6025
6035 CLOSE #0
6040 IF n >3 THEN n=n-3
6045 RETURN
```

The subroutine to display the WAVES.TXT file on the LCD display clears the screen, then skips over lines in the file that have 'scrolled off' the top of the display contained in the b variable. It then 'prints' the next 8 lines which is all that the LCD can show at a time. The LCD print subroutine t$ variable receives the file name from each line, discarding the following file size information. The t variable receives the desired display print type, 1=normal, 2=inverse depending upon whether or not the screen row index variable i matches the current screen selection variable s, and the currently selected .WAV filename is saved in variable s$. Finally the LCDPRINT rows variable r receives the computed row bit number and the line is printed on the LCD display:

```
7000 REM
7001 REM Display file list on LCD
7002 REM
7005 LCDCLEAR 255,0,127:OPEN #0,"WAVES.TXT","r"
7010 FOR i=0 TO b:INPUT #0,l$:NEXT i
7015 FOR i=0 TO 7
7020 INPUT #0,l$:t$=LEFT$(l$,FIND(l$," "))
7025 t=1:LIF i=s THEN t=2:s$=t$
7030 r=1<<i:LCDPRINT r,0,4,t,0,t$
7035 NEXT i
7040 CLOSE #0
7045 RETURN
```

The @MSG$ event handler subroutine is called whenever a delimited message string has been received from the LCD display. The handler captures the received @MSG$ into the k$ variable, freeing the special variable to receive another message. The received message is then parsed to see if a LCD Reset message or Keypress message has been received. Display reset messages simply refresh the display. Keypress messages are further decoded to determine which key was pressed on the display and are dispatched to corresponding code fragments for processing.

Currently, only 3 keys are handled; the Up and Down arrows, and the Enter key between them. The Down arrow key advances the selection variable s to the bottom of the display, then advances the display skip lines variable b as required, redrawing the display. The Up arrow key decrements the selection variable s to the top of the display, then decrements the display skip lines variable b as required, redrawing the display. The Enter key stops any currently queued sound that is playing and starts the selected sound playing:

```
8000 REM
8001 REM LCD received message handler
8002 REM
8005 k$=@MSG$
8010 IF MID$(K$,0,1) ="K" THEN 8050
8015 IF MID$(K$,0,1) ="R" THEN 8025
8020 RETURN
8024 REM R command
8025 GOSUB 7005
8030 RETURN
8049 REM K commands
8050 k=ASC(MID$(K$,2,1)) -ASC("0")
8055 ON k,GOTO 8100,8200,8300,8400,8500,8600,8700
8060 RETURN
8099 REM K30 - left most key
8100 RETURN
8199 REM K31 - mid left key
```

```
8200 RETURN
8299 REM K32 - mid right key
8300 RETURN
8399 REM K33 - right most key
8400 RETURN
8499 REM K34 - down arrow key
8500 LIF ((s<n) &(s<7)) THEN s=s+1:GOTO 8510
8505 IF ((n>s)&((b+s)<(n-2))) THEN b=b+1
8510 GOSUB 7005
8515 RETURN
8599 REM K35 - up arrow key
8600 LIF S>0 THEN s=s-1:GOTO 8610
8605 IF b>0 THEN b=b-1
8610 GOSUB 7005
8615 RETURN
8699 REM K36 - enter key
8700 @SOUND$="":@SOUND$=s$:RETURN
```

Running the program while connected to the PC with Hyperterminal using the above cable produces the following text. Notice the ACS-LCD-128x64 commands delimited with the ASCII SOH (01) / ETX (03) characters:

```
run
 CFF007F  CFF007F  P0100420TWO.WAV  P0200410THREE.WAV  P0400410ONE.WAV  P0800410FIVE.WAV
  P1000410SIX.WAV  P2000410SEVEN.WAV  P4000410EIGHT.WAV  P8000410NINE.WAV
```

Renaming the program to CFSOUND.BAS and placing it along with the requisite sound files onto the CF card will turn your CFSound-3 into a Jukebox with LCD display. Here's the entire program:

```
1 REM
2 REM LCD Jukebox Demo
3 REM
10 REM clear display, install @msg$ handler
15 LCDCLEAR 255,0,127
25 ONEVENT @MSG$,GOSUB 8005
30 REM generate and display wav file list
35 GOSUB 6005:GOSUB 7005
100 a=0
110 GOTO 100
6000 REM
6001 REM Generate list of .WAV files
6002 REM
6005 OPEN #0,"WAVES.TXT","w"
6010 DIR #0,*.wav
6015 CLOSE #0
6020 n=0:OPEN #0,"WAVES.TXT","r"
6025 INPUT #0,l$
6030 LIF LEN(l$) >0 THEN n=n+1:GOTO 6025
6035 CLOSE #0
6040 IF n >3 THEN n=n-3
6045 RETURN
7000 REM
7001 REM Display file list on LCD
7002 REM
7005 LCDCLEAR 255,0,127:OPEN #0,"WAVES.TXT","r"
7010 FOR i=0 TO b:INPUT #0,l$:NEXT i
7015 FOR i=0 TO 7
7020 INPUT #0,l$:t$=LEFT$(l$,FIND(l$," "))
7025 t=1:LIF i=s THEN t=2:s$=t$
7030 r=1<<i:LCDPRINT r,0,4,t,0,t$
7035 NEXT i
7040 CLOSE #0
7045 RETURN
8000 REM
8001 REM LCD received message handler
8002 REM
8005 k$=@MSG$
8010 IF MID$(K$,0,1) ="K" THEN 8050
8015 IF MID$(K$,0,1) ="R" THEN 8025
8020 RETURN
8024 REM R command
8025 GOSUB 7005
8030 RETURN
8049 REM K commands
```

```
8050 k=ASC(MID$(K$,2,1)) -ASC("0")
8055 ON k,GOTO 8100,8200,8300,8400,8500,8600,8700
8060 RETURN
8099 REM K30 - left most key
8100 RETURN
8199 REM K31 - mid left key
8200 RETURN
8299 REM K32 - mid right key
8300 RETURN
8399 REM K33 - right most key
8400 RETURN
8499 REM K34 - down arrow key
8500 LIF ((s<n) &(s<7)) THEN s=s+1:GOTO 8510
8505 IF ((n>s)&((b+s)<(n-2))) THEN b=b+1
8510 GOSUB 7005
8515 RETURN
8599 REM K35 - up arrow key
8600 LIF S>0 THEN s=s-1:GOTO 8610
8605 IF b>0 THEN b=b-1
8610 GOSUB 7005
8615 RETURN
8699 REM K36 - enter key
8700 @SOUND$="":@SOUND$=s$:RETURN
```

Fixed Length Record File I/O

Here's a short demonstration of the FOPEN, FREAD and FWRITE commands:

```
5 DEL "test.dat"
10 FOPEN #1,20,"test.dat"
15 INPUT "how many records:",n
20 FOR r=0 TO n-1
30 FWRITE #1,r,r,"str"+STR$(r)
40 NEXT r
50 PRINT "reading records..."
60 r=0
70 FREAD #1,r,b,b$
75 IF @FEOF(#1) THEN 1000
80 PRINT "rec:",r,"=",b,",",b$
90 r=r+1:GOTO 70
1000 CLOSE #1
Ready
run
how many records:10
reading records...
rec: 0= 0,str0
rec: 1= 1,str1
rec: 2= 2,str2
rec: 3= 3,str3
rec: 4= 4,str4
rec: 5= 5,str5
rec: 6= 6,str6
rec: 7= 7,str7
rec: 8= 8,str8
rec: 9= 9,str9
Ready
type test.dat
0,"str0"
1,"str1"
2,"str2"
3,"str3"
4,"str4"
5,"str5"
6,"str6"
7,"str7"
8,"str8"
9,"str9"
Ready
```

Error Logging

While developing programs without a serial connection, or for stand alone program monitoring it may be advantageous to record any program errors that occur to the CF card. Then when the program stops running, the CF card can be inserted into a PC card reader and the error that caused the program to stop can be examined. The following code sets up ONERROR to transfer control to line 32000 where an ERRORS.TXT file is opened for appended writing and the causal error message is written at the end of the file:

```
10 REM Error Logging Example
20 ONERROR GOTO 32000
30 A=B/0
32000 OPEN #0,"ERRORS.TXT","a+w"
32005 PRINT #0,ERR$()
32010 CLOSE #0
32015 STOP
Ready
run
STOP in line 32015
Ready
type errors.txt
Divide by zero error in line 30
Ready
run
STOP in line 32015
Ready
type errors.txt
Divide by zero error in line 30
Divide by zero error in line 30
Ready
```

DMX Control Synchronized to Sound

This example plays an audio file for an exhibit at the Alamo Museum in San Antonio, Texas. The CFSound-III with DMX module synchronizes the fading up/down of the house lights and scene lights with the audio track.

Here's how it works. The show is started by pressing a button connected to the Contact #25 input. The show stops by pressing a button connected to Contact #26 or when the show's sound file ends.

The @SOUNDFRAMEPRESCALER specialvar is set to 50. This causes a @SOUNDFRAMESYNC event to fire every second while the sound is playing. The subroutine at line 1000 is executed every time this happens and uses the one second sound frame number to start DMX channels fading up/down to make the show happen.

```
10 REM Program to fade DMX controlled lamps up and down during the playout of audio file
15 REM Start DMX
20 @DMXMASTER=1:@SOUNDFRAMEPRESCALER=50
25 REM Stop Show
30 @SOUND$="":ONEVENT @SOUNDFRAMESYNC,GOSUB 0:GOSUB 9000
35 REM Check for show start button
40 IF @CLOSURE(24)=0 THEN 40
42 @CLOSURE(24)=0
45 REM Show start
50 ONEVENT @SOUNDFRAMESYNC,GOSUB 1000
55 @SOUND$="ALAMO.WAV"
60 REM Check for show end (sound or button)
65 IF (@CLOSURE(25)=0) AND (@SOUND$<>"") THEN 65
70 @CLOSURE(25)=0:@SOUND$=""
75 GOTO 25
1000 REM Sound Frame Sync handler
1005 S=@SOUNDFRAMESYNC
1010 REM Phil's Intro
1015 LIF S=1 THEN C1=0:I1=255:M1=127:GOSUB 10100:RETURN
1020 REM Charli
1025 LIF S=170 THEN C1=0:I1=127:M1=0:GOSUB 10100:C0=1:I0=0:M0=255:GOSUB 10000:RETURN
1030 REM Lunette
1035 LIF S=185 THEN C1=1:I1=255:M1=0:GOSUB 10100:C0=2:I0=0:M0=255:GOSUB 10000:RETURN
1040 REM Bowie's room
1045 LIF S=220 THEN C1=2:I1=255:M1=0:GOSUB 10100:C0=3:I0=0:M0=255:GOSUB 10000:RETURN
1050 REM Kitchen
1055 LIF S=243 THEN C1=3:I1=255:M1=0:GOSUB 10100:C0=4:I0=0:M0=255:GOSUB 10000:RETURN
1060 REM Ramp
1065 LIF S=249 THEN C1=4:I1=255:M1=0:GOSUB 10100:C0=5:I0=0:M0=255:GOSUB 10000:RETURN
1070 REM Gunade
1075 LIF S=281 THEN C1=5:I1=255:M1=0:GOSUB 10100:C0=6:I0=0:M0=255:GOSUB 10000:RETURN
1080 REM Trevino
1085 LIF S=295 THEN C1=6:I1=255:M1=0:GOSUB 10100:C0=7:I0=0:M0=255:GOSUB 10000:RETURN
1090 REM XCastenada
1095 LIF S=313 THEN C1=7:I1=255:M1=0:GOSUB 10100:C0=8:I0=0:M0=255:GOSUB 10000:RETURN
1100 REM norCasten
1105 LIF S=326 THEN C1=8:I1=255:M1=0:GOSUB 10100:C0=9:I0=0:M0=255:GOSUB 10000:RETURN
1110 REM Teran
1115 LIF S=350 THEN C1=9:I1=255:M1=0:GOSUB 10100:C0=10:I0=0:M0=255:GOSUB 10000:RETURN
1120 REM Long Barracks
1125 LIF S=371 THEN C1=10:I1=255:M1=0:GOSUB 10100:C0=11:I0=0:M0=255:GOSUB 10000:RETURN
1130 REM convent
1135 LIF S=402 THEN C1=11:I1=255:M1=0:GOSUB 10100:C0=12:I0=0:M0=255:GOSUB 10000:RETURN
1140 REM ConventCourt
1145 LIF S=429 THEN C1=12:I1=255:M1=0:GOSUB 10100:C0=13:I0=0:M0=255:GOSUB 10000:RETURN
1150 REM SouthCourt
1155 LIF S=438 THEN C1=13:I1=255:M1=0:GOSUB 10100:C0=14:I0=0:M0=255:GOSUB 10000:RETURN
1160 REM Fortin de Cos
1165 LIF S=482 THEN C1=14:I1=255:M1=0:GOSUB 10100:C0=15:I0=0:M0=255:GOSUB 10000:RETURN
1170 REM moonlight
1175 LIF S=503 THEN C1=15:I1=255:M1=0:GOSUB 10100:C0=16:I0=0:M0=255:GOSUB 10000:RETURN
1180 REM 4th column
1185 LIF S=540 THEN C1=16:I1=255:M1=0:GOSUB 10100:C0=17:I0=0:M0=255:GOSUB 10000:RETURN
1190 REM 1st 2nd columns
1195 LIF S=557 THEN C1=17:I1=255:M1=0:GOSUB 10100:C0=18:I0=0:M0=255:GOSUB 10000:RETURN
1200 REM 3rd column
1205 LIF S=565 THEN C1=18:I1=255:M1=0:GOSUB 10100:C0=19:I0=0:M0=255:GOSUB 10000:RETURN
1210 REM Foothold Nor
```

```
1215 LIF S=588 THEN C1=19:I1=255:M1=0:GOSUB 10100:C0=20:I0=0:M0=255:GOSUB 10000:RETURN
1220 REM Low Barrack
1225 LIF S=604 THEN C1=20:I1=255:M1=0:GOSUB 10100:C0=21:I0=0:M0=255:GOSUB 10000:RETURN
1230 REM Long Barracks
1235 LIF S=611 THEN C1=21:I1=255:M1=0:GOSUB 10100:C0=22:I0=0:M0=255:GOSUB 10000:RETURN
1240 REM Convent On
1245 LIF S=612 THEN C0=23:I0=0:M0=255:GOSUB 10000:RETURN
1250 REM Long Barracks Off
1255 LIF S=623 THEN C1=22:I1=255:M1=0:GOSUB 10100:RETURN
1260 REM Bowie's Room On
1265 LIF S=642 THEN C0=24:I0=0:M0=255:GOSUB 10000:RETURN
1270 REM Convent Off
1275 LIF S=676 THEN C1=23:I0=0:M0=255:GOSUB 10100:RETURN
1280 REM Bowie's Room Off
1285 LIF S=645 THEN C1=24:I1=255:M1=0:GOSUB 10100:RETURN
1290 REM Palisade On
1295 LIF S=681 THEN C0=25:I0=0:M0=255:GOSUB 10000:RETURN
1300 REM Palisade Off
1305 LIF S=686 THEN C1=25:I1=255:M1=0:GOSUB 10100:RETURN
1310 REM Church On
1315 LIF S=688 THEN C0=26:I0=0:M0=255:GOSUB 10000:RETURN
1320 REM Dawn
1325 LIF S=696 THEN GOSUB 11000:RETURN
1330 REM 1 thru 8 off
1335 LIF S=710 THEN S1=1:E1=7:GOSUB 11100:RETURN
1340 REM 9 thru 14 off
1345 LIF S=717 THEN S1=8:E1=13:GOSUB 11100:RETURN
1350 REM 15 thru 20 off
1355 LIF S=726 THEN S1=14:E1=19:GOSUB 11100:RETURN
1360 REM 21 thru 27 off
1365 LIF S=735 THEN S1=20:E1=26:GOSUB 11100:RETURN
1370 RETURN
9000 REM Fadeup house lights, others off
9005 C0=0:I0=@DMXDATA(0):M0=255:GOSUB 10000
9010 FOR C9=1 TO 31:@DMXDATA(C9)=0:NEXT C9
9015 RETURN
10000 REM Fadeup channel C0 from I0 to M0
10005 ONEVENT @TIMER(0),GOSUB 10050
10010 F0=I0:@TIMER(0)=2
10015 RETURN
10050 IF F0<=(M0-4) THEN F0=F0+4 ELSE F0=M0
10055 @DMXDATA(C0)=F0
10060 LIF F0<>M0 THEN @TIMER(0)=2:RETURN
10065 ONEVENT @TIMER(0),GOSUB 0:RETURN
10100 REM Fadedown channel C1 from I1 to M1
10105 ONEVENT @TIMER(1),GOSUB 10150
10110 F1=I1:@TIMER(1)=2
10115 RETURN
10150 IF F1>=(M1+4) THEN F1=F1-4 ELSE F1=M1
10155 @DMXDATA(C1)=F1
10160 LIF F1<>M1 THEN @TIMER(1)=2:RETURN
10165 ONEVENT @TIMER(1),GOSUB 0:RETURN
11000 REM Fadeup all channels except house
11005 ONEVENT @TIMER(0),GOSUB 11050
11010 F0=0:@TIMER(0)=2
11015 RETURN
11050 IF F0<=(255-4) THEN F0=F0+4 ELSE F0=255
11055 FOR C9=1 TO 31:@DMXDATA(C9)=F0:NEXT C9
11060 LIF F0<>255 THEN @TIMER(0)=2:RETURN
11065 ONEVENT @TIMER(0),GOSUB 0:RETURN
11100 REM Fadedown channels S1->E1
11105 ONEVENT @TIMER(1),GOSUB 11150
11110 F1=255:@TIMER(1)=2
11115 RETURN
11150 IF F1>(0+4) THEN F1=F1-4 ELSE F1=0
11155 FOR C9=S1 TO E1:@DMXDATA(C9)=F1:NEXT C9
11160 LIF F1<>0 THEN @TIMER(1)=2:RETURN
11165 ONEVENT @TIMER(1),GOSUB 0:RETURN
```

Firmware Revisions

Version	Date	Notes
1.0	5-17-02	First started development.
1.1	10-20-04	Changes to run on CFSound-III prototype.
1.2	8-11-06	Additions to allow sound playing.
1.3	8-29-06	Changed DisplayProgramListing() to add a preceding space to a secondary keyword if it's preceded by an unsubscripted specialvar.
1.4	9-20-06	Upgrade VDSP toolset from 3.5 to 4.5. Changed MEMORY specialvar to call new heap_space_unused(0) to show program memory left. Added @BAUD special var. Added MULDIV() function. Added support for string lexicographical relation checking with <,<=,>,>= operators. Added divide by zero checking on /, % and MULDIV function. Increased size of available program memory from 4095 bytes to 131068 bytes by moving the heap from L1 to L2 memory. Added FIND() function. Corrected @MSG$ variable events. Corrected MID$() index to be zero based. Added LCDx statements to support ACS-LCD-128x64 on serial port.
1.5	11-15-06	Added @PTT special var. Clear CFSound Red LED indicator flashes if RUN command issued.
1.6	11-29-06	Added LCDBITMAP command.
1.7	2-08-07	Added @MUTE special var. Un-mute amplifier if RUN command issued.
1.8	6-25-07	Added @PORT2 special var and support functions for new CFSound-3 revision 3.
1.9	7-31-07	Added @LINEIN special var. Disable line input if RUN command issued. Added @NSVOL special var that changes the current volume but doesn't save it to NVRAM. Corrected syntax error on attempts to access @PORT2 special var.
1.10	8-29-07	Corrected LCDBITMAP command memory free() calls to be in reverse matching order to calloc() calls to minimize memory fragmentation. Increased size of the available program memory from 131068 to 524284 bytes. Cleared any pending TIMER events upon TIMER assignment. Clear pending escapes when RUN command issued. Fix SYNTAX_ERROR on empty command line. Fixed problem with WAIT statement hanging up due to ONEVENT handling clearing events between statements - now only clear events between statements if there is an event handler defined and executed. Added EDIT line command. Corrected @YEAR to return two digit year. Corrected the ability to GOTO self AND still process events.
1.11	9-12-07	Corrected @CONTACT()= assignments to be active true (non-zero assignment turns output contact on). Corrected ABS() function to return correct value. Rewrote string handling to be to be similar to numeric expression handling allowing true nesting. Change @DOW to be read-only, 0=Sunday -> 6=Saturday computed from the epoch Thursday January 1, 1970. Added VAL() to return the numeric value of a string argument. Added STR$() to return the string representation of a numeric argument. Added optional ELSE clause to IF/THEN statement. Added FOPEN #N, FREAD #N and FWRITE #N commands. Added @FEOF(#N) specialvar. Removed useless EXIT command. Added EXITFOR command to allow exiting to a line from within a for/next loop without receiving a nesting error. Added AND / OR logical operators. Corrected operator priority so that statements like A=0 OR A=2 and A*2+3 evaluate correctly. Changed bitwise ! to ~. Added NOT operator. Added ERR$() function to return string representation of last error number. Added FINSERT #N and FDELETE #N commands. Corrected DEL, REN and MD commands to allow use in programs. Added INSERT$ and REPLACE$ commands. Corrected LIF to return syntax error if line # appears after THEN.
1.12	9-24-07	Conditionalized contact closure and opening processing in BasicTimer_Process() to only set the event and remove the closure if there is an event handler defined to allow the use of @CLOSURE and @OPENING in a program without an event handler.
1.13	10-30-07	Increased stream I/O buffer size from 512 to 32256 bytes to speed up program loading and file I/O statements. Added escape detection to terminate TYPE command output. Added MULMOD() function.
1.14	11-27-07	Corrected @PORT() and @PORT2() special variables to update the OutputContacts[] as well if the port number is <=2.
1.15	4-22-08	Added call to flush uart tx queue in Basic_Process if escape sequence is detected to interrupt long program & type command output. Added @DMXxxxx specialvars. Fixed bug in ORDER statement not finding the referenced DATA statement.
1.16	4-22-08	Added @DMXANALOG specialvars. Corrected a race condition that caused @SOUND$ events to be missed.
1.17	8-27-08	Corrected lockup bug with FIND() function. Corrected Syntax Error in INPUT #N statements when end of file #N is reached. Added FINPUT #N

Version	Date	Notes
		statement. Corrected FREAD and FINPUT statements to give Out of Data error if they run out of data in the file #N before all of the variables are assigned values. Added three new error codes. Added @LCDADDRESS specialvar. Added additional error descriptions following the line number for some Syntax errors. Added @LCDTYPE specialvar to affect operation of LCDx statements and provide support for ACS-LCD320x240 on serial port.
1.18	12-09-08	Internal development version for DMX testing.
1.19	2-10-09	Added @SOUNDFRAMEPRESCALER and @SOUNDFRAMESYNC specialvars. Fixed syntax error display of "Expected 'x'" to correct the display of anticipated keyword tokens.
1.20	2-25-09	Added delay after setting @DMXRESET to allow DMX CPU time to reset.

NOTICE:

This Information or any portion thereof remains the property of ACS. The Information contained herein is believed to be accurate and ACS assumes no responsibility or liability for its use in any way and conveys no license or title under any patent or copyright and makes no representation or warranty that this Information is free from patent or copyright infringement.

ACS PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS WITHOUT THE EXPRESS WRITTEN APPROVAL OF ACS.

As used herein:

1. Life support devices or systems are devices or systems which, (a) are intended for surgical implant into the body, or (b) support or sustain life, and whose failure to perform, when properly used in accordance with instructions for use provided in the labeling, can be reasonably expected to result in a significant injury to the user.

2. A critical component is any component of a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness.